AD-A239 712



ATION PAGE

Form Approved
OPM No. 0704-0188

Publi
needi
Head
Mana

er response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data
den estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington
son Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 11 Feb 1991 to 01Jun 1993

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Cross Developm,ent System, Version 4.1, for VAX/VMS to 68k, MicroVAX 3800(Host) to Motorola MVME 133A-20 (MC68020)(Target). 910121l1.11124

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 087

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

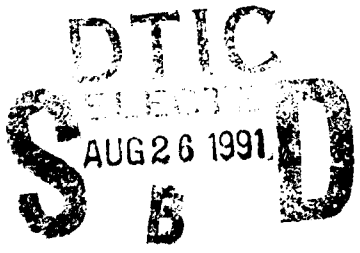
10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
TeleSoft, TeleGen2 Ada Cross Development System, Version 4.1, Ottobrunn, Germany, for VAX/VMS to 68k, MicroVAX 3800 (under VAX/VMS Version V5.2))(Host) to Motorola MVME 133A-20 (MC68020)(bare machine)(Target), ACVC 1.11.



91-08750



14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev 2-89)
Prescribed by ANSI Std. 239-128

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-01-21.

Compiler Name and Version: **TeleGen2™ Ada Cross Development System, Version 4.1, for VAX/VMS to 68k**

Host Computer System: **MicroVAX 3800 (under VAX/VMS Version V5.2)**

Target Computer System: **Motorola MVME 133A-20 (MC68020) (bare machine)**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #910121I1.11124 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

DTIC COPY INSPECTED 6

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist Special
A-1

AVF Control Number: IABG-VSR 087
11 February 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 910121I1.11124
TeleSoft
TeleGen2™ Ada Cross Development System
Version 4.1, for VAX/VMS to 68k
MicroVAX 3800 =>
Motorola MVME 133A-20 (MC68020)

== based on TEMPLATE Version 91-01-10 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-01-21.

Compiler Name and Version:	**TeleGen2™ Ada Cross Development System, Version 4.1, for VAX/VMS to 68k**
Host Computer System:	**MicroVAX 3800 (under VAX/VMS Version V5.2)**
Target Computer System:	**Motorola MVME 133A-20 (MC68020) (bare machine)**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #910121I1.11124 is awarded to TeleSoft. This certificate expires on 01 March 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer:	TeleSoft 5959 Cornerstone Court West San Diego CA USA 92121
Ada Validation Facility:	IABG, Dept. ITE W-8012 Ottobrunn Germany
ACVC Version:	1.11

Ada Implementation:

Ada Compiler Name and Version:	TeleGen2™ Ada Cross Development System, Version 4.1, for VAX/VMS to 68K
Host Computer System:	MicroVAX 3800 (under VAX/VMS Version V5.2)
Target Computer System:	Motorola MVME 133A-20 (MC68020) (bare machine)

Customer's Declaration

I, the undersigned, declare that TeleSoft has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.



Date: 1/23/91

TELESOFT
Raymond A. Parra
Vice President
General Counsel

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 USE OF THIS VALIDATION SUMMARY REPORT 1-1
1.2 REFERENCES 1-2
1.3 ACVC TEST CLASSES 1-2
1.4 DEFINITION OF TERMS 1-3

CHAPTER 2 IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS 2-1
2.2 INAPPLICABLE TESTS 2-1
2.3 TEST MODIFICATIONS 2-4

CHAPTER 3 PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT 3-1
3.2 SUMMARY OF TEST RESULTS 3-2
3.3 TEST EXECUTION 3-2

APPENDIX A MACRO PARAMETERS

APPENDIX B COMPILATION SYSTEM OPTIONS

APPENDIX C APPENDIX F OF THE Ada STANDARD

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 91-01-09.

E28005C	B28006C	C34006D	C35702A	B41308B	C43004A
C45114A	C45346A	C45612B	C45651A	C46022A	B49008A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
B83026B	C83026A	C83041A	B85001L	C97116A	C98003B
BA2011A	CB7001A	CB7001B	CB7004A	CC1223A	BC1226A
CC1226B	BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A
CD2A21E	CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A
CD2B15C	BD3006A	BD4008A	CD4022A	CD4022D	CD4024B
CD4024C	CD4024D	CD4031A	CD4051D	CD5111A	CD7004C
ED7005D	CD7005E	AD7006A	CD7006E	AD7201A	AD7201E
CD7204B	AD7206A	BD8002A	BD8004C	CD9005A	CD9005B
CDA201E	CE2107I	CE2117A	CE2117B	CE2119B	CE2205B
CE2405A	CE3111C	CE3116A	CE3118A	CE3411B	CE3412B
CE3607B	CE3607C	CE3607D	CE3812A	CE3814A	CE3902B

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C35508I..J and C35508M..N (4 tests) include enumeration representation clauses for Boolean types in which the specified values are other than (FALSE => 0, TRUE => 1); this implementation does not support a change in representation for Boolean types. (See section 2.3.)

C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types; for this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C, CA2009F, BC3204C, and BC3205D check whether a generic unit can be instantiated BEFORE its generic body (and any of its subunits) is compiled. This implementation creates a dependence on generic units as allowed by AI-00408 and AI-00530 such that the compilation of the generic unit bodies makes the instantiating units obsolete. (See section 2.3)

LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests) check for pragma INLINE for procedures and functions.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

CE2107B..E (4 tests), CE2107L, CE2110B, and CE2111D attempt to associate multiple internal files with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

CE2107G..H (2 tests), CE2110D, and CE2111H attempt to associate multiple internal files with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A attempt to associate multiple internal files with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page

number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 26 tests.

C35508I..J and C35508M..N (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests attempt to change the representation of a boolean type. The AVO ruled that, in consideration of the particular nature of boolean types and the operations that are defined for the type and for arrays of the type, a change of representation need not be supported; the ARG will address this issue in Commentary AI-00564.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

BA1001A	BA2001C	BA2001E	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A	

CA2009C, CA2009F, BC3204C, and BC3205D were graded inapplicable by Evaluation Modification as directed by the AVO. Because the implementation makes the units with instantiations obsolete (see section 2.2), the Class C tests were rejected at link time and the Class B tests were compiled without error.

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A22J, CD2A24A, and CD2A31A..C (3 tests) use instantiations of the support procedure Length_Check, which uses Unchecked_Conversion according to the interpretation given in AI-00590.
The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instantiations of Length_Check--i.e., the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for both technical and sales information about this Ada implementation system, see:

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121, USA
(619) 457-2700

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a)	Total Number of Applicable Tests	3801	
b)	Total Number of Withdrawn Tests	84	
c)	Processed Inapplicable Tests	84	
d)	Non-Processed I/O Tests	0	
e)	Non-Processed Floating-Point Precision Tests	201	
f)	Total Number of Inapplicable Tests	285	(c+d+e)
g)	Total Number of Tests for ACVC 1.11	4170	(a+b+f)

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 285 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by a serial communications link, and run. The results were captured on the host computer system.

Test output, compiler and linker listings, and job logs were captured on a magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test are given on the next page, which was supplied by the customer.

Compiler Option Information

B TESTS:

TSADA/E68/ADA/LIST='testname'/VIRT=3000 'testfile'

option	description
E68	choose VAX/E68 compiler
ADA	invoke TeleGen2 Ada cross compiler
LIST=< >	generated interspersed error listing
VIRTUAL_SPACE	set virtual space of library manager other than default (2500)
'testfile'	the filename being compiled

NON_B TESTS:

VME133 board:

TSADA/E68/ADA/VIRTUAL=3000 'testfile'
TSADA/E68/LINK/LOAD_MODULE='testname'/OPTIONS=<additional opt> 'mainname'

option	description
E68	choose VAX/E68 compiler
ADA	invoke TeleGen2 Ada cross compiler
VIRTUAL_SPACE	set virtual space of library manager other than default (2500)
LINK	invoke TeleGen2 Ada linker
LOAD_MODULE=< >	specify name of executable created
OPTIONS=<additional opt>	use additional options from the named options file
'testfile'	the filename being compiled
'mainname'	the name of the main compilation unit

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	"CCCCCCC10CCCCCCCC20CCCCCCCC30CCCCCCCC40 CCCCCCCC50CCCCCCCC60CCCCCCCC70CCCCCCCC80 CCCCCCCC90CCCCCCC100CCCCCCC110CCCCCCC120 CCCCCCC130CCCCCCC140CCCCCCC150CCCCCCC160 CCCCCCC170CCCCCCC180CCCCCCC190CCCCCC199"

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	TELEGEN2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.40283E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	4.25354E+37
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	0.0
$HIGH_PRIORITY	63

```
$ILLEGAL_EXTERNAL_FILE_NAME1
                        BADCHAR*^/%

$ILLEGAL_EXTERNAL_FILE_NAME2
                        /NONAME/DIRECTORY

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.ADA")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006D1.ADA")

$INTEGER_FIRST          -32768

$INTEGER_LAST           32767

$INTEGER_LAST_PLUS_1    32768

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        MCI'(OP => NOP);

$MACHINE_CODE_TYPE      Opcodes

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   NO_SUCH_TYPE_AVAILABLE

$NAME_LIST              TELEGEN2

$NAME_SPECIFICATION1    SYS$WATCHER:[VME133]X2120A.;1

$NAME_SPECIFICATION2    SYS$WATCHER:[VME133]X2120B.;1
```

$NAME_SPECIFICATION3	SYS$WATCHER:[VME133]X3119A.;1
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_SYS_NAME	TELEGEN2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	RECORD NULL; END RECORD;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	2048
$TICK	0.01
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2

APPENDIX B

COMPILATION SYSTEM AND LINKER OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

TELESOFT

TeleGen2 Ada Development System for VAX®/VMS® Systems to Embedded MC680X0 Targets

Compiler Command Options

OPT-1745N-V1.1(VAX.E68) 21JAN91

Version 4.01

Copyright © 1991, TeleSoft.
All rights reserved.

Copyright © 1991, TeleSoft. All rights reserved.
TeleSoft® is a registered trademark of TeleSoft.
TeleGen2™ is a trademark of TeleSoft.
VAX® and VMS® are registered trademarks of Digital Equipment Corp.®.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at DFAR 252.227-7013, or FAR 52.227-14, ALT III and/or FAR 52.227-19 as set forth in the applicable Government Contract.

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

1 **Compiler command options** 1-1
1.1 ADA 1-1
1.2 BIND 1-17
1.3 LINK 1-22

1
Compiler command options

1.1. ADA

[Compiler]

Introduction

After access to TeleGen2 has been established and a library has been created, you can invoke the Ada compiler via the ADA command. The general command format for compilation is

```
TSADA/E68/ADA/[<qualifier>[,...]] <file>
```

where

`<qualifier>` One of the qualifiers available for the compiler.

`<file>` One in a possible series of file specifications, separated by commas, indicating the unit(s) to be compiled. If /INPUT_LIST is used, <file> is interpreted as a file containing a list of files to be compiled. The default source file type is .ADA, and the default list file type is .LIS. A file name may be qualified with a location in standard VAX/VMS format. A source or input list file may reside on any directory in the system.

The default qualifier settings are designed to allow for the simplest and most convenient use of the compiler. For most applications, no additional qualifier setting need be specified. However, optional qualifiers are provided to perform special functions.

Qualifiers

/ABORT_COUNT=<n>
/ABORT_COUNT=999 (default)

The /ABORT_COUNT qualifier is an execution control qualifier for the compiler. It allows you to set the maximum number of errors the compiler can locate before it aborts. This qualifier can be used with any combination of compiler options. The minimum value is 1 and the default value is 999.

The compiler maintains separate counts of all syntactic errors, semantic errors, and warning messages issued during a compilation. If any of these counts becomes too great, you may want to abort the compilation. If the compiler does find errors, you can abort the compilation by entering control-Y (^Y) or you can wait until the checking is completed. The /ABORT_COUNT qualifier allows you to determine the number of errors that you believe is reasonable before the compiler aborts.

/ASSEMBLY_CODE[=<file>]
/NOASSEMBLY_CODE (default)

With the /ASSEMBLY_CODE qualifier, you can obtain an assembly listing of compiler generated code for a unit or a collection. You can use this qualifier with the compiler, the binder and the optimizer in the same way. The listing is similar to that produced with the /MACHINE_CODE qualifier, except that it will not contain location or offset information. The file produced with the /ASSEMBLY_CODE qualifier is suitable as input to an assembler. In contrast, the file produced by /MACHINE_CODE is more suitable for human readability. The default file name is the same as it would be for /MACHINE_CODE, i.e., the unit name being compiled.

/ASSEMBLY_CODE and /MACHINE_CODE are mutually exclusive.

/BIND=<main>[<qualifier>]
/NOBIND (default)

This qualifier, when used with the compiler, enables you to bind a main program more efficiently by combining the binding process with the compilation process. This feature is especially useful when compiling single-unit test programs. The binding of more than one main program is not supported, though programs that contain multiple compilation units may be bound.

Default settings for the qualifier values were chosen for the simplest and most convenient use of the binder. For most applications, no additional qualifiers are required. Optional qualifiers are provided, as shown in the following list:

/DEBUG	Bind the program for use with the debugger.
/PROFILE	Bind the program for use with the profiler.
/LIBFILE	Specify a library other than the default LIBLST.ALB.
/TEMPLIB	Specify a library other than the default LIBLST.ALB.
/SHOW_TASK_EXCEPTION	Cause unhandled exceptions in tasks to be reported in the same manner as those that occur in the main program.
/TASK_STACK_SIZE	Set the default amount of stack to allocate from the Ada heap for each task.
/STACK_GUARD_SIZE	Specify the amount of additional stack space to allocate (in addition to /TASK_STACK_SIZE) for each task.
/TRACEBACK	Set the depth of the run-time exception traceback report.

/CONTEXT=<n>
/CONTEXT=1 (default)

When an error message is output, it is helpful to include the lines of the source program that surround the line containing the error. These lines provide a context for the error in the source program and help to clarify the nature of the error. The /CONTEXT qualifier controls the number of source lines that immediately precede and follow the error. The qualifier affects the error messages output on SYS$OUTPUT or in a listing. In a mixed listing, the context lines will be duplicated. In a listing using /NOMIXED, the context lines will be included for each error listed after the complete source listing. The default setting for this qualifier is 1.

Note: The main program unit must be located in the working sublibrary. If this is not the case, reorder the sublibraries in the library file or use TSADA/E68/MOVE or TSADA/E68/COPY to move or copy the unit to the working sublibrary.

If these conditions are met, you may proceed to bind and link the program.

/DEBUG
/NODEBUG (default)

To use the debugger, you must compile, bind, and link program units using the /DEBUG qualifier. This ensures that source debugging information and a link map are put into the Ada library for use by the debugger. The /DEBUG qualifier causes the compiler to save High Form for debugging purposes. It also causes the compiler to generate Debugging Information (DI) for any unit that is to be used with the debugger.

Because this qualifier is positional, the debug information is generated only for the specified files. For example,

```
$ TSADA/E68/ADA A,B/DEBUG,C
```

compiles A.ADA and B.ADA without debugging information, and compiles C.ADA with debugging information. For information on positional qualifiers, see the *Overview*.

The default setting of this qualifier is /NODEBUG. The use of /DEBUG ensures that the High Form and debugger information for secondary units are not deleted. While the compilation time overhead generated by use of /DEBUG is minimal, retaining this optional information in the Ada Library increases the space overhead. To check if a compilation unit has been compiled using /DEBUG, use the TSADA/E68/SHOW/EXTENDED command for the unit.

/DIAGNOSTICS[=<file>]
/NODIAGNOSTICS (default)

If warnings or errors are encountered during compilation, this qualifier specifies that the compiler or binder is to produce a diagnostics file (with the file type .DIA). This file contains information that allows you to use the VAX Language Sensitive Editor (LSE) to quickly correct source errors. More information can be found in the LSE manual.

This qualifier is positional, so the location of the command and its parameters on the command line is important. Only diagnostics files for the specified files are generated.

/ENABLE_TRACEBACK
/NOENABLE_TRACEBACK (default)

In the unlikely event that you should receive an Unexpected Error Condition message, you should contact Customer Support. Customer Support may request that you provide additional information about the error condition by including the /ENABLE_TRACEBACK qualifier in the compiler invocation that fails. This qualifier allows the tool to display the exception traceback associated with the unexpected error condition. The information provided in the traceback will allow Customer Support to diagnose the problem more efficiently.

/GRAPH[=<file>]
/NOGRAPH (default)

The /GRAPH qualifier is used with the optimizer while compiling, and it generates a textual representation call graph for the unit being compiled and optimized. This qualifier is positional, so the location of the command and its parameters on the command line is important. The specified parameters for /GRAPH are the graph files to be generated. For instance, the command

```
TSADA/E68/ADA A,B/GRAPH, C
```

compiles A.ADA, then sends a graph to B.GRF, then compiles C.ADA. For information on positional qualifiers, see the *Overview*.

The default setting is /NOGRAPH. If you specify graph, the following file specifications can result:

`<file>`	A name you specify for the file to which the generated graph is sent.
`<unit>.GRF`	A name provided for the file by default, where <unit> is the name of the unit being compiled when optimizing during compilation.

If multiple units are being compiled and optimized, and call graphs are desired, /GRAPH should be used without specifying a file name. The graph for each unit will be located in a separate file named after the unit. If you specify a <file>, separate versions of the file will be created for each unit.

/INPUT_LIST
/NOINPUT_LIST (default)

As a default, the file list specified in the TSADA/E68 command is the list of specifications of the files containing the Ada units to be compiled. When you specify the /INPUT_LIST qualifier, the compiler assumes that the command contains one file specification, and that this file contains a list of source files to be compiled. When this qualifier is used, the input file should contain the source file specifications, one per line. For example, to compile source files CALC_ARITH.ADA and CALC_MEM.ADA in the current default directory and file CALC_IO.ADA in directory [CIOSRC], you could first prepare a file CALC_COMPILE.LIS containing the following text:

```
CALC_ARITH
CALC_MEM
[CIOSRC]CALC_IO
```

The command:

```
$ TSADA/E68/ADA/INPUT_LIST  CALC_COMPILE.LIS
```

would then cause these three files to be compiled in sequence.

In addition to the names of the source files, the input list may contain comments in Ada syntax, i.e., all text on a line including and following the comment marker "--" will be ignored.

/INPUT_LIST is a positional qualifier. The specified parameters are input lists, and all other parameters are source files. For example, the command line

```
$ TSADA/E68/ADA A,B/INPUT_LIST,C
```

compiles A.ADA first. It then compiles all files in B.LIS, and finally, it compiles C.ADA.

/INPUT_LIST has the same advantages and effects as a multi-file compilation specified on the command line. /INPUT_LIST is useful when a long list of files to compile has been readily obtained from a directory listing or a TSADA/E68/SHOW library report.

When the /INPUT_LIST qualifier is used in conjunction with the default /UPDATE qualifier, the working sublibrary is updated after each unit that successfully compiles. If a unit in the list fails to compile due to errors, the sublibrary is still updated for all other successfully compiled units before and after it in the list. If /NOUPDATE is used, the sublibrary is not updated for any unit if one unit fails to compile.

/INPUT_LIST
/NOINPUT_LIST (default)

As a default, the file list specified in the TSADA/E68 command is the list of specifications of the files containing the Ada units to be compiled. When you specify the /INPUT_LIST qualifier, the compiler assumes that the command contains one file specification, and that this file contains a list of source files to be compiled. When this qualifier is used, the input file should contain the source file specifications, one per line. For example, to compile source files CALC_ARITH.ADA and CALC_MEM.ADA in the current default directory and file CALC_IO.ADA in directory [CIOSRC], you could first prepare a file CALC_COMPILE.LIS containing the following text:

```
CALC_ARITH
CALC_MEM
[CIOSRC]CALC_IO
```

The command:

```
$ TSADA/E68/ADA/INPUT_LIST  CALC_COMPILE.LIS
```

would then cause these three files to be compiled in sequence.

In addition to the names of the source files, the input list may contain comments in Ada syntax, i.e., all text on a line including and following the comment marker "--" will be ignored.

/INPUT_LIST is a positional qualifier. The specified parameters are input lists, and all other parameters are source files. For example, the command line

```
$ TSADA/E68/ADA A,B/INPUT_LIST,C
```

compiles A.ADA first. It then compiles all files in B.LIS, and finally, it compiles C.ADA.

/INPUT_LIST has the same advantages and effects as a multi-file compilation specified on the command line. /INPUT_LIST is useful when a long list of files to compile has been readily obtained from a directory listing or a TSADA/E68/SHOW library report.

When the /INPUT_LIST qualifier is used in conjunction with the default /UPDATE qualifier, the working sublibrary is updated after each unit that successfully compiles. If a unit in the list fails to compile due to errors, the sublibrary is still updated for all other successfully compiled units before and after it in the list. If /NOUPDATE is used, the sublibrary is not updated for any unit if one unit fails to compile.

If the /BIND qualifier is used in conjunction with the /INPUT_LIST qualifier, the main program unit name may be given as the value of the /BIND qualifier to identify which unit in the list is the main program. If not specified, the last unit of the input list is assumed to be the main program.

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

By default, the library file named LIBLST with a default type of .ALB is used by the TeleGen2 tool set to determine which set of sublibraries are to be referenced during the operation of the tool. This file must be present in the working directory. With the /LIBFILE qualifier, you can specify a library other than the default, LIBLST.ALB. /TEMPLIB may also be used to create an alternative library. However, the /TEMPLIB and /LIBFILE qualifiers are mutually exclusive; only one or the other qualifier may be used at the same time.

When you specify the /LIBFILE qualifier, you indicate the file specification of an alternative library file that contains the list of sublibraries and optional comments. If you do not specify a file type with the file name, the system uses the file type .ALB.

For example, consider a library file named WORKLIB.ALB with the contents:

```
Name: MYWORK
Name: [CALCPROJ]CALCLIB
Name: TSADA$E68:[LIB.68020]RTL
```

You could specify

```
$ TSADA/E68/ADA/LIBFILE=WORKLIB
```

As an alternative to using /LIBFILE, you may assign the library file specification to the logical name LIBLST. For example,

```
$ ASSIGN WORKLIB.ALB LIBLST
```

/LIST=<file>
/NOLIST=<file> (default)

The /LIST qualifier to the /ADA compiler command produces a file containing a source listing with numbered lines and any error messages. This qualifier is positional, so the location of the qualifier and its parameters on the command line is important. A list will only be generated for the specified parameters.

The compiler always outputs error messages to the device specified by SYSSOUTPUT. The LIST qualifier causes the error listing to be

incorporated in the source listing file as well. The default listing contains the errors intermixed with the source code.

You can provide a file specification to the /LIST qualifier which indicates the VMS file to receive the error output.

The default is for /LIST is /NOLIST. The /NOLIST qualifier will suppress error output to the listing file.

If the /LIST qualifier is used without a file specification, the error output is sent to the file named <file>.LIS, where <file> is the name of the source file being compiled. If you provide a file specification, the output is sent to the specified file instead of <file>.LIS. If your file specification does not include a file type, the system gives the file the type, .LIS.

If a file name is specified and multiple source files are being compiled, the listing for each file is output to a separate version of the file name specified.

/MACHINE_CODE[=<file>]
/NOMACHINE_CODE (default)

The /MACHINE_CODE qualifier allows you to obtain an assembly listing of the code that the compiler generates for a unit or a collection. The listing consists of assembly code intermixed with source code as comments. Note that the listing generated by this qualifier is independent of the source/error listing generated by the /LIST qualifier. The default for this qualifier is /NOMACHINE_CODE.

The listing output is sent to a file named <unit>_.S if the unit is a library unit, and <unit>.S if the unit is a secondary unit. <unit> is the name of the compilation unit that is being listed .

If multiple compilation units are being compiled and you have provided a file specification, the machine code listing for each compilation unit will be output to a different version of the same file name.

If the compilation unit name is longer than 39 characters, the name will be truncated at 39 characters. No listing will be generated if there are syntactic or semantic errors in the compilation.

/MONITOR
/NOMONITOR (default)

Normally, the only visible output produced by the TeleGen2 tool set during operation is error or warning messages. The /MONITOR qualifier enables the reporting of version numbers and messages that allow you to monitor the tool's progress. When you specify /MONITOR, the output is sent to standard output (SYS$OUTPUT).

/OBJECT (default)
/NOOBJECT

The /NOOBJECT qualifier instructs the compiler to perform syntactic and semantic analysis of the source program without generating object code. /NOOBJECT sets a default of /NOSQUEEZE to ensure that the High Form and Low Form are preserved for secondary units. The default setting is /OBJECT, which allows the generation of object code.

/OPTIMIZE[=<option>[,...]][<qualifier>]
/NOOPTIMIZE (default)

The /OPTIMIZE qualifier causes the compiler to invoke the optimizer to optimize the Low Form generated by the middle pass for the unit being compiled. The code generator takes the optimized Low Form as input and produces more efficient object code.

/NOOPTIMIZE is the default. The /NOOPTIMIZE qualifier gives the quickest compilation turnaround, but does not perform many code optimizations. This results in code that may run slower and be larger than normal. Code intended for use with the Debugger must be compiled at this level of optimization.

This qualifier is positional, so the location of the qualifier and its parameters is important. /OPTIMIZE will only optimize the specified parameters. For example, the qualifier may appear on both on the verb that is changing the default and on one or more parameters. The verb qualifier indicates that all parameters will be optimized. The parameter qualifier indicates that Y is not to be optimized.

```
$ TSADA/E68/ADA/OPTIMIZE X,Y/NOOPTIMIZE,Z
```

The result is that the files in X.ADA are compiled and optimized, followed by the file Y.ADA. The files in Z.ADA are compiled, but not optimized.

There are two parameters that are used with /OPTIMIZE. These are:

`<option>` Zero or more of optimizer options on the command line, separated by commas.

`<qualifier>` The qualifier /NOGRAPH or /GRAPH[=<file>] /NOGRAPH is the default value. /GRAPH generates a call graph for the unit being compiled. The default output file is <unit>.GRF when you do note specify a <file>. (See the Global Optimizer for more information on this qualifier.)

You can specify zero or more optimizer options to control optimization. These options are listed here briefly. For more information on these options, see the Optimize command in the *Command Summary*.

Options	Defaults	Operation
ALL \| NONE \| SAFE	ALL	Enable/disable certain optimizations, or permit only safe optimizations.
[NO]AUTOINLINE [NO]INLINE[:<file>]	AUTOINLINE INLINE	Controls automatic inlining. Enables inline expansion of subprograms.
[NO]PARALLEL	PARALLEL	Subprograms may be called from parallel tasks.
[NO]RECURSE	RECURSE	Subprograms may be called recursively.
[NO]SPEED	SPEED	Produce fastest code but slowest compilation speed.

Depending on the amount of optimization you require, you can operate the compiler on one of three optimization levels. You would typically use /NOOPTIMIZE during debugging and development, while levels OPTIMIZE=NOSPEED or OPTIMIZE=SPEED would be used for final product development. The SPEED option instructs TeleGen2 to produce the fastest executable code, even at the expense of a slower compilation.

/PROFILE
/NOPROFILE (default)

The /PROFILE qualifier causes the code generation phase of compilation to place special profiler run-time code into the generated object module. This qualifier is a positional qualifier when it is used with /ADA. /PROFILE generates profile information only for specified files because of this characteristic. For information on positional qualifiers, see the *Overview*. For example, /PROFILE can take the following actions, depending on its location on the command line.

`. . ./ADA/PROFILE A,B,C`	produces profile info for A, B and C.
`. . ./ADA A,B/PROFILE,C`	produces profile info only for C.
`. . ./ADA/PROFILE A,B/NOPROFILE,C`	produces profile info only for A and C.

/PROFILE can be used with the /OPTIMIZE option, which causes the code generator to include the profiler run-time code following the optimization. If you use the /BIND qualifier, /PROFILE instructs the binder to link in the profiler run-time support routines. If you have compiled any code in a program with the /PROFILE qualifier, then you must also supply the /PROFILE qualifier to the binder if the program is bound separately.

The default for this qualifier is /NOPROFILE.

/SQUEEZE (default)
/NOSQUEEZE

When you compile an Ada program, the compiler stores two intermediate code representations of the program in the library. These code representations are known as High Form and Low Form. High Form must be retained for a library unit because it is required for the compilation of any units that reference it. For example, a compiled package specification's High Form are used by the corresponding package body when it is compiled. However, intermediate forms of a secondary unit, such as a package body, may frequently be discarded after its compilation. Discarding this information results in a significant decrease in library size (typically 50 to 70 percent for multi-unit programs).

The /SQUEEZE qualifier can be used with the compiler or the optimizer (/OPTIMIZE). Using the /SQUEEZE qualifier during compilation causes the intermediate forms to be discarded after compilation, if possible. /NOSQUEEZE causes the full intermediate forms to be saved in all cases.

Note: The optimizer (/OPTIMIZE), and cross-referencer (/XREF) programs require unsqueezed units. If you are going to use one of these programs, you must compile the units using /NOSQUEEZE.

The default for this qualifier is /SQUEEZE, with one exception. This is the /NOOBJECT qualifier which is commonly used when compiling units for collective optimization. In this case, the object code is not required, but unsqueezed units are. Thus, use of the /NOOBJECT qualifier also causes /NOSQUEEZE to be the default. In either of these cases, use of an explicit /SQUEEZE or /NOSQUEEZE qualifier overrides the default.

To verify whether or not a unit has been squeezed, use the TSADA/E68/SHOW/EXTENDED command for the unit. A unit has not been squeezed if and only if the attributes High_Form and Low_Form appear in the listing for that unit.

/SUPPRESS[=<option>[,...]]
/NOSUPPRESS (default)

The /SUPPRESS qualifier allows you to suppress selected run-time checks and/or source line references in generated object code.

The Ada language requires, as a default, a wide variety of run-time checks to ensure the validity of operations. For example, arithmetic overflow checks are required on all numeric operations, and range checks are required on all assignment statements that could result in an illegal value being assigned to a variable. While these checks are vital during development and an important asset of the language, they introduce a substantial overhead. This overhead may be prohibitive in time-critical applications. Thus, the Ada language provides a way to selectively suppress classes of checks via the Suppress pragma. However, use of the pragma requires modifications to the Ada source.

The /SUPPRESS qualifier provides a functional alternative to the Suppress pragma. /SUPPRESS allows you to suppress checks in the compiler invocation command without modifying the source code. The Suppress pragma is valid in any declarative region of a package and affects all nested regions. The /SUPPRESS qualifier is equivalent to adding pragma Suppress to the beginning of the declarative part of each compilation unit in a file.

The compiler also stores source line and subprogram name information by default in the object code. This information is used to display a source level traceback when an unhandled exception propagates to the outer level of a program. This information is also particularly valuable during development as it provides a direct

indication of the source line at which the exception occurs and the subprogram calling chain that led to the line generating the exception.

The source line information introduces an overhead of 6 bytes for each line of source that causes code to be generated. Thus, a 1000-line package may have up to 6000 bytes of source information. For one compilation unit, the extra overhead (in bytes), is the total length of all subprogram names in the unit (including Middle Pass generated subprograms), plus the length of the compilation unit name. For certain space-critical applications, this extra space may be unacceptable and may be inhibited with the /SUPPRESS qualifier. When the source line information is inhibited, the traceback indicates the offset of the object code at which the exception occurs, instead of the source line number. When the subprogram name information is inhibited, the traceback indicates the offsets of the subprogram calls in the calling chain, instead of the subprogram names.

When you specify an <option>, it represents one of a possible list of options separated by commas. These options indicate the features to be suppressed. The default setting is /NOSUPPRESS.

The options and their actions are presented in the following table. The names of the options may be abbreviated as long as they remain unique within the set of options. All options except SOURCE_INFO and *ALL function as if a corresponding Suppress pragma were present in* the Ada source. The exception is that /SUPPRESS=(ELABORATION_CHECK) differs from pragma Suppress(Elaboration_Check). The switch suppresses elaboration checks made by other units on this unit. The pragma suppresses elaboration checks made on other units from this unit. The NAME_INFO option specifies that subprogram name information is to be suppressed in the object code. The SOURCE_INFO option specifies that source line information is to be suppressed in the object code. The ALL_CHECKS option suppresses all run-time checks listed in the table. The ALL option specifies that subprogram name information, source line information, and all run-time checks in the table are to be suppressed.

For example, the qualifier:

```
/SUPPRESS=(SOURCE,ELAB)  MY_FILE
```

inhibits the generation of source line information and elaboration checks in the object code of the units in file MY_FILE.

ALL	Suppress source line information and all run-time checks listed below.
NONE	Equivalent to /NOSUPPRESS
SOURCE_INFO	Suppress source line information in object code.
NAME_INFO	Suppress subprogram name information in object code.
ALL_CHECKS	Suppress all access checks, discriminant checks, division checks, elaboration checks, index checks, length checks, overflow checks, range checks, and storage checks.
ELABORATION_CHECK	Suppress all elaboration checks.
OVERFLOW_CHECK	Suppress all overflow checks.
STORAGE_CHECK	Suppress all storage checks.

/TASK_STACK_SIZE=<n>
/TASK_STACK_SIZE=4096 (default)

The /TASK_STACK_SIZE qualifier sets the default amount of stack to allocate from the Ada heap for each task. The <n> you specify is the size of the task stack in bytes.

/TEMPLIB=<sublib>[,...]
/LIBFILE=LIBLST.ALB (default)

This qualifier allows you to define a temporary library consisting of a selection of sublibraries. The temporary library may be used for the duration of a single command. In all uses, the /TEMPLIB and /LIBFILE options are mutually exclusive; only one or the other qualifier may be used at the same time.

<sublib> The name of the sublibrary, optionally prefixed with the specification of the VMS directory in which it resides. If no directory is specified, the current default directory is assumed. Multiple sublibrary file specifications are separated by commas in a list.

Semantically, the argument string of this qualifier is the logical equivalent of a library file containing the listed sublibraries, one per

line, in the order listed. Thus, we could list the sublibraries:

```
/TEMPLIB=(MYWORK,[CALCPROJ]CALCLIB,TSADA$E68:[LIB.68020]RTL)
```

/TIME_SLICE_QUANTUM=<n>
/TIME_SLICE_QUANTUM=0 (default)

This qualifier specifies the slice of time, in milliseconds, in which a task is allowed to execute before the run-time switches control to another ready task of equal priority. This timeslicing activity allows for periodic round-robin scheduling among equal-priority tasks. Timeslicing may or may not be implemented for a particular environment.

The default value for /TIME_SLICE_QUANTUM is 0 (i.e., timeslicing is disabled). No run-time overhead is incurred when timeslicing is disabled.

/UPDATE (default)
/[NO]UPDATE

When multiple source files are being compiled, the /UPDATE qualifier instructs the compiler to update the library after each source file is compiled. The default setting is /UPDATE.

If /NOUPDATE is used, and an error occurs during compilation, the working sublibrary is not updated at all, for any unit, even for remaining units in the source file in error. All remaining source files will be compiled for syntactic and semantic errors only. The /NOUPDATE qualifier is advantageous to use when it is known that all the source files will compile without error and the user wishes to save the overhead time involved in updating the library for each source file.

/VIRTUAL_SPACE=<n>
/VIRTUAL_SPACE=10000 (default)

This qualifier specifies the number of 1kb pages that will be used in memory while the tool executes. Greater values will usually improve performance, but will result in more physical memory requirements.

1.2. BIND

[Binder]

Introduction

The object code files generated by the compiler are TeleGen2-defined Object Form files stored in the Ada library. These files must be bound to create a linkable object. The binder program generates the code needed to elaborate the components in a consistent order. The binder is invoked when you use the /BIND qualifier.

The general command format of the bind step is

```
TSADA/E68/BIND[<qualifier>,...] <main>
```

where

`<main>`	indicates the name of the unit to be used as the main program
`<qualifier>`	none or more binder qualifiers

The following diagram illustrates how to put qualifiers and parameters together into commands in order to invoke the binder.

Qualifiers

/ASSEMBLY_CODE[=<file>]
/NOASSEMBLY_CODE (default)

With the /ASSEMBLY_CODE qualifier, you can obtain an assembly listing of compiler generated code for a unit or a collection. You can use this qualifier with the compiler, the binder and the optimizer in the same way. The listing is similar to that produced with the /MACHINE_CODE qualifier, except that it will not contain location or offset information. The file produced with the /ASSEMBLY_CODE qualifier is suitable as input to an assembler. In contrast, the file produced by /MACHINE_CODE is more suitable for human readability. The default file name is the same as it would be for /MACHINE_CODE, i.e., the unit name being compiled.

/ASSEMBLY_CODE and /MACHINE_CODE are mutually exclusive.

/DEBUG
/NODEBUG (default)

To use the debugger, you must compile, bind, and link program units using the /DEBUG qualifier. This ensures that source debugging information and a link map are put into the Ada library for use by the debugger. The /DEBUG qualifier causes the binder to save the intermediate forms of code for debugging purposes.

The default setting of this qualifier is /NODEBUG. The use of /DEBUG ensures that the High Form and debugger information for secondary units are not deleted. While the compilation time overhead generated by use of /DEBUG is minimal, retaining this optional information in the Ada Library increases the space overhead. To check if a compilation unit has been bound using /DEBUG, use the TSADA/E68/SHOW/EXTENDED command for the unit.

/ENABLE_TRACEBACK
/NOENABLE_TRACEBACK (default)

In the unlikely event that you should receive an Unexpected Error Condition message, you should contact Customer Support. Customer Support may request that you provide additional information about the error condition by including the /ENABLE_TRACEBACK qualifier in the compiler invocation that fails. This qualifier allows the tool to display the exception traceback associated with the unexpected error condition. The information provided in the traceback will allow Customer Support to diagnose the problem more efficiently.

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

By default, the library file named LIBLST with a default type of .ALB is used by the TeleGen2 tool set to determine which set of sublibraries are to be referenced during the operation of the tool. This file must be present in the working directory. With the /LIBFILE qualifier, you can specify a library other than the default, LIBLST.ALB. /TEMPLIB may also be used to create an alternative library. However, the /TEMPLIB and /LIBFILE qualifiers are mutually exclusive; only one or the other qualifier may be used at the same time.

When you specify the /LIBFILE qualifier, you indicate the file specification of an alternative library file that contains the list of sublibraries and optional comments. If you do not specify a file type with the file name, the system uses the file type .ALB.

For example, consider a library file named WORKLIB.ALB with the contents:

```
Name: MYWORK
Name: [CALCPROJ]CALCLIB
Name: TSADA$E68:[LIB.68020]RTL
```

You could specify

```
$ TSADA/E68/ADA/LIBFILE=WORKLIB
```

As an alternative to using /LIBFILE, you may assign the library file specification to the logical name LIBLST. For example,

```
$ ASSIGN WORKLIB.ALB LIBLST
```

/MACHINE_CODE[=<file>]
/NOMACHINE_CODE (default)

The /MACHINE_CODE qualifier allows you to obtain an assembly listing of the code that the compiler generates for a unit or a collection. The listing consists of assembly code intermixed with source code as comments. Note that the listing generated by this qualifier is independent of the source/error listing generated by the /LIST qualifier. The default for this qualifier is /NOMACHINE_CODE.

The listing output is sent to a file named <unit>_.S if the unit is a library unit, and <unit>.S if the unit is a secondary unit. <unit> is the name of the compilation unit that is being listed .

If multiple compilation units are being compiled and you have provided a file specification, the machine code listing for each compilation unit will be output to a different version of the same file name.

If the compilation unit name is longer than 39 characters, the name will be truncated at 39 characters. No listing will be generated if there are syntactic or semantic errors in the compilation.

/MONITOR
/NOMONITOR (default)

Normally, the only visible output produced by the TeleGen2 tool set during operation is error or warning messages. The /MONITOR qualifier enables the reporting of version numbers and messages that allow you to monitor the tool's progress. When you specify /MONITOR, the output is sent to standard output (SYS$OUTPUT).

/PROFILE
/NOPROFILE (default)

If you are binding a previously compiled program, the /PROFILE qualifier causes the binder to enable the elaboration calls for the target program to be profiled.

Note: If one or more of the units to be bound have been compiled with the /PROFILE qualifier, then you must supply this qualifier to the binder.

/TASK_STACK_SIZE = <n>
/TASK_STACK_SIZE = 4096 (default)

The /TASK_STACK_SIZE qualifier sets the default amount of stack to allocate from the Ada heap for each task. The <n> you specify is the size of the task stack in bytes.

/TEMPLIB = <sublib> [,...]
/LIBFILE = LIBLST.ALB (default)

This qualifier allows you to define a temporary library consisting of a selection of sublibraries. The temporary library may be used for the duration of a single command. In all uses, the /TEMPLIB and /LIBFILE options are mutually exclusive; only one or the other qualifier may be used at the same time.

`<sublib>` The name of the sublibrary, optionally prefixed with the specification of the VMS directory in which it resides. If no directory is specified, the current default directory is assumed. Multiple sublibrary file specifications are separated by commas in a list.

Semantically, the argument string of this qualifier is the logical equivalent of a library file containing the listed sublibraries, one per line, in the order listed. Thus, we could list the sublibraries:

```
/TEMPLIB=(MYWORK,[CALCPROJ]CALCLIB,TSADA$E68:[LIB.68020]RTL)
```

The /TEMPLIB qualifier applies to both compilation and binding, so it need be specified only once and may appear in any order on the command line when you use BIND. The binder needs to have present every compilation unit referenced by the main program. If a unit is missing, the binder will report the error and will not be invoked. Therefore, you should be sure that the set of sublibraries specified by the /TEMPLIB qualifier contains all the units belonging to the main program.

/TIME_SLICE_QUANTUM=<n>
/TIME_SLICE_QUANTUM=0 (default)

This qualifier specifies the slice of time, in milliseconds, in which a task is allowed to execute before the run-time switches control to another ready task of equal priority. This timeslicing activity allows for periodic round-robin scheduling among equal-priority tasks. Timeslicing may or may not be implemented for a particular environment.

The default value for /TIME_SLICE_QUANTUM is 0 (i.e., timeslicing is disabled). No run-time overhead is incurred when timeslicing is disabled.

/VIRTUAL_SPACE=<n>
/VIRTUAL_SPACE=5000 (default)

This qualifier specifies the number of 1kb pages that will be used in memory while the tool executes. Greater values will usually improve performance, but will result in more physical memory requirements.

1.3. LINK

[Linker]

Introduction

The Ada Linker is a component of the TeleGen2 system that allows you to link compiled Ada programs in preparation for target execution. The linker resolves references within the Ada program, the bare target run-time support library, and any imported non-Ada object code. To support the development of embedded applications, the linker is designed to operate in a variety of modes and to handle many types of output format.

The linker links together OF modules to construct executable load modules. (See the "Linker" chapter in the *User Guide* for details). Optionally, the linker outputs symbol location information that is used by the debugger. The linker can also output information used by the profiler. All unused subprograms will be eliminated from the executable image.

The command syntax for the Ada Linker is:

```
$ TSADA/E68/LINK[<qualifier>] [<unit>]
```

where

`<qualifier>` None or more of the command line qualifiers available with the linker.

`<unit>` An optional command line parameter indicating the name of the Ada compilation unit to be linked as a main program.

Not that the compilation unit must have been bound as a main program prior to linking. If you do not provide the unit name on the command line, then the unit is specified using the INPUT option in an options file.

Linker directives are communicated to the linker as qualifiers on the VMS command line or as options entered via an options file or SYS$INPUT. Command line qualifiers are useful for controlling options that you are likely to change often. The default qualifier settings are designed to allow for the simplest and most convenient use of the linker.

Command line qualifiers and parameters enable you to:

- Specify the name and format of the linked output file
- Control the generation and format of listing map files
- Specify an options file
- Specify the starting memory location for the linked output
- Specify the library file containing the components to be linked
- Control the output of debug symbol information for debugging
- Monitor the linking process
- Profile the linking process

More complicated linker options, such as the specification of memory locations for specific portions of the code or data for a program, are input via options in a linker options file. Linker options may be used to:

- Specify the compilation units to be used as input to the Linker, the library search paths, and the usage of the input files
- Specify the name and format of the linked output file
- Control the generation and format of listing map files produced by the Linker.
- Specify the location of named memory regions and reserved memory regions in physical memory
- Specify the location of control sections in physical memory
- Define symbol values
- Specify the target machine on which the output is to be executed

The linker qualifiers are illustrated in the diagram following this discussion. Each of the /LINK qualifiers is described in detail in the Qualifier section. Following the qualifier descriptions, there is a detailed discussion on linker options files and their qualifiers.

Qualifiers

/BASE = <address>
/BASE = 0 (default)

This qualifier is used with the linker to specify the start location of the linked output. The linker locates non-absolute control sections in consecutive memory locations. All control sections are word aligned on the MC680X0. The <address> is a valid MC680X0 address. You can specify the address as an unsigned octal (%Ooctal), decimal (%Ddecimal), or hexadecimal (%Xhexadecimal) value in VMS format. The default is hexadecimal.

If you specify neither the /BASE qualifier nor an options file LOCATE and the link is complete, the Linker uses the default location value of address 0.

The /BASE qualifier governs the location for any code, constant, or data section not covered by an options file LOCATE. This qualifier does not supercede any LOCATE options. The /BASE qualifier is equivalent to a LOCATE option with no control section or component name specified.

/DEBUG
/NODEBUG (default)

To use the debugger, you must compile, bind, and link program units using the /DEBUG qualifier. This ensures that source debugging information and a link map are put into the Ada library for use by the debugger. This qualifier controls the generation of debug symbol information for use with the debugger. The information is in the form of a link map that associates machine addresses with the symbol names found in a compilation unit. The debugger uses the link map to locate the address of the beginning of a compilation unit and the addresses of source lines and link names.

A program that you want to run with the debugger must be linked with the /DEBUG option. If supported by the chosen load module format, /DEBUG may also cause symbol information to be output in the load module. The qualifier is ignored if you select /OBJECT_FORM. In the standard configuration of the TeleGen2 system, none of the outputs support symbol information in the load module. The default is /NODEBUG.

/ENABLE_TRACEBACK
/NOENABLE_TRACEBACK (default)

In the unlikely event that you should receive an Unexpected Error Condition message, you should contact Customer Support. Customer Support may request that you provide additional information about the error condition by including the /ENABLE_TRACEBACK qualifier in the compiler invocation that fails. This qualifier allows the tool to display the exception traceback associated with the unexpected error condition. The information provided in the traceback will allow Customer Support to diagnose the problem more efficiently.

/EXCLUDED
/NOEXCLUDED (default)

The /EXCLUDED qualifier is used with the linker to insert a list of excluded subprograms into the link map listing. The default is /NOEXCLUDED.

/EXCLUDED is also a subqualifier of the /MAP linker qualifier. The /MAP qualifier controls the generation and format of the listing map files that the linker produces. With the /EXCLUDED subqualifier, the /MAP qualifier generates a section of the link map that lists Ada subprograms that have been excluded from the linked object file. These subprograms were excluded because they were not used in the call graph of the main program that is being linked.

/EXECUTE_FORM (default)

This qualifier is used with the linker (TSADA/E68/LINK). It is used to specify that the load module output of the linker should be Execute Form. Execute Form is the default output format generated by the Linker and is suitable for use as input to the download and receiver utilities. If you use /EXECUTE_FORM, it must immediately follow TSADA/E68/LINK on the command line.

The load module produced has the file type .EF.

/IMAGE
/NOIMAGE (default)

The /IMAGE qualifier is used with the /MAP qualifier or MAP option in the linker. /IMAGE generates a memory image listing in addition to the link map listing generated by /MAP. The linker writes the image listing to the same file as the link map listing. This is the only optional section of the listing. The default is /NOIMAGE.

In a memory image listing, each nonsequential section of the image in memory starts on a new page. The image listing contains the locations in memory that are printed as hexadecimal values. Each line in the listing is filled with the amount of data on a line that is a multiple of 16 bytes and up to the specified or default WIDTH limit.

Relocatable control sections are printed with a location that is relative to the start of the control section.

/LIBFILE=<file>
/LIBFILE=LIBLST.ALB (default)

By default, the library file named LIBLST with a default type of .ALB is used by the TeleGen2 tool set to determine which set of sublibraries are to be referenced during the operation of the tool. This file must be present in the working directory. With the /LIBFILE qualifier, you can specify a library other than the default, LIBLST.ALB. /TEMPLIB may also be used to create an alternative library. However, the /TEMPLIB and /LIBFILE qualifiers are mutually exclusive; only one or the other qualifier may be used at the same time.

When you specify the /LIBFILE qualifier, you indicate the file specification of an alternative library file that contains the list of sublibraries and optional comments. If you do not specify a file type with the file name, the system uses the file type .ALB.

For example, consider a library file named WORKLIB.ALB with the contents:

```
Name: MYWORK
Name: [CALCPROJ]CALCLIB
Name: TSADA$E68:[LIB.68020]RTL
```

You could specify

```
$ TSADA/E68/ADA/LIBFILE=WORKLIB
```

As an alternative to using /LIBFILE, you may assign the library file specification to the logical name LIBLST. For example,

```
$ ASSIGN WORKLIB.ALB LIBLST
```

/LINES_PER_PAGE=<n>
/LINES_PER_PAGE=66 (default)

With the /LINK command, /LINES_PER_PAGE specifies the number of lines per listing page. The default is 66. You may specify a positive integer greater than 10.

/LOAD_MODULE[=<file>]
/LOAD_MODULE=<main>.EF (default)

This qualifier is used with the linker to specify the VMS file name for the load module output created by the linker.

The <file> is the optional VMS file specification for the output. If <file> does not include an file type, the linker will append a file type appropriate to the chosen load module format (.SR, .EF, .MEM, .I3E for /SRECORDS, /EXECUTE_FORM, and /IEEE, respectively). If you do not specify an output file, the linker writes the linked output to:

```
<main>.<type>
```

The <main> is the Ada name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a valid VMS file specification. The <type> is the appropriate file type for the selected load module format.

You can use the /LOAD_MODULE qualifier with the /OPTIONS qualifier. Any output file specification present in the options file is superceded by the specification on the command line. If the /LOAD_MODULE qualifier is used with the /OBJECT_FORM qualifier, both formats will be produced.

/LOCALS
/NOLOCALS (default)

This qualifier includes local symbols in the link map symbol listing.

/MAP[=<file>]
/NOMAP (default)

This qualifier is used to request and control a link map listing. The format of the link map listing file is described in the *User Guide*.

When you specify this option, you can optionally list a <file>. This is the optional VMS file specification for the output. If you do not specify a file type, the Linker uses a default file type of .MAP. If you do not specify an output file, the Linker writes the listing to

```
<unit>.MAP
```

where

`<unit>` Represents the name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option, modified as necessary to form a

valid VMS file specification.

You control the output of the MAP qualifier using one or more of the following qualifiers:

/[NO]IMAGE
/[NO]LOCALS
/[NO]EXCLUDED
/WIDTH = <132 | 80>
/LINES_PER_PAGE = <50 | n>

/IMAGE generates a memory image listing in addition to the map listing. The Linker writes the image listing to the same file as the link map listing. This is the only optional section of the listing. The default is /NOIMAGE.

/LOCALS includes local symbols in the link map symbol listing. The default is /NOLOCALS.

/EXCLUDED inserts a list of excluded subprograms into the link map listing. The default is /NOEXCLUDED.

/WIDTH specifies the width of the lines in the listing file. The default value is 132 characters. The alternate width is 80 characters.

/LINES_PER_PAGE specifies the number of lines per listing page. The default is 50. You may specify a positive integer greater than 10.

A command line /MAP qualifier supercedes any MAP options in an options file. The default is /NOMAP. /NOMAP can be used on the command line to suppress MAP options specified in an options file.

/MONITOR
/NOMONITOR (default)

Normally, the only visible output produced by the TeleGen2 tool set during operation is error or warning messages. The /MONITOR qualifier enables the reporting of version numbers and messages that allow you to monitor the tool's progress. When you specify /MONITOR, the output is sent to standard output (SYS$OUTPUT).

/OBJECT_FORM[= <lib_comp>]
/NOOBJECT_FORM (default)

This qualifier specifies that one output of the linker is to be linked OF. Linked OF is suitable for incomplete modules and can be used subsequently as input to the Ada linker. The linked OF is put into the library as an object form module (OFM) component.

<lib_comp> Represents a library component name.

<unit> The Ada name of the main program unit (if present), the name specified as the command line parameter, or the name specified as the first INPUT option. The output of the link is put into the library as the object form module called <unit> if you have not specified a library component name.

Note that the object form module of <unit> is a library component separate from that of the specification or body of the unit.

If an object form module library component with the specified name already exists in the current working sublibrary, that component is deleted and replaced by the new output.

The /OBJECT_FORM qualifier may be used with the /OPTIONS qualifier. Any format or name present in the options file is superceded by the format and name specified on the command line. You may request /OBJECT_FORM instead of the default /EXECUTE_FORM, or in addition to a load module format. To obtain both an object form module and a load module, you must enter both qualifiers.

/OPTIONS[=<file>]
/NOOPTIONS (default)

/OPTIONS specifies that the linker is to process additional options obtained interactively, or from a linker options file.

<file> This is a valid VMS file specification. It represents a file that contains linker options. If no file type is present, the linker uses the default file type .OPT.

The default file specification is SYS$INPUT. The default is /NOOPTIONS.

The /LOAD_MODULE qualifier and the /OBJECT_FORM qualifier may be used with the /OPTIONS qualifier. Any format present in the options file is superceded by the format specified on the command line when you use /LOAD_MODULE with /OPTIONS. When /OBJECT_FORM is used with /OPTIONS, any output file specification present in the options file is superceded by the specification on the command line.

/PROFILE
/NOPROFILE (default)

/PROFILE should be used if you want to profile the program you are linking. When you specify /PROFILE, the linker generates a subprogram dictionary for the program. This dictionary is later used as input by the profiler. The dictionary is a text file containing the name and addresses of all subprograms in the program. The name of the dictionary file defaults to

```
<load_mod>.DIC
```

To link your program for profiling, you enter the link command, followed by the linker options file specification, the /PROFILE qualifier and the name of the load module:

```
TSADA/E68/LINK/OPTIONS=<file>/PROFILE <main>
```

The linker options file is described in the "Linker" chapter in the *User Guide*.

You must also include the run-time profiling support in the target program. This is done by modifying the options file to include the profiling version of the environment module and the prelinked profiler support, CGS_PROFILE. The profiler support also references the user-written Receiver Serial_IO support module, *SIOxxx*, as described in the *VME133 Target Programmer's User Guide*.

A profiling options file therefore differs from a non-profiling options file by the specification of different OFM modules. A non-profiling options file would contain the following /INPUT specifications:

```
INPUT/OFM ENVxxx     !Include environment for machine 'xxx'
--
--
INPUT/MAIN MY_PROGRAM !Include user program
! etc.
```

the corresponding profiling version would contain:

```
INPUT/OFM ENVxxx_PROFILE  !Include profiling environment for machine 'xxx'
INPUT/OFM CGS_PROFILE     !Include prelinked profiling CGS
INPUT/OFM xxxSIO          !Include Serial_Io support for machine 'xxx'
--
--
INPUT/MAIN MY_PROGRAM !Include user program
! etc.
```

Instead of linking in the ENVxxx_PROFILE, CGS_PROFILE, and xxxSIO modules individually, you may create a pre-linked OF module containing all three modules and place it under the name xxx_PROFILE_SYSTEM in the Ada library. The program options file would then be:

```
INPUT/OFM xxx_PROFILE_SYSTEM  !Include profiling support for machine '
--
--
INPUT/MAIN MY_PROGRAM !Include user program
!etc.
```

/SRECORDS
/EXECUTE_FORM (default)

The /SRECORDS qualifier is used with the linker to specify that the output of the Linker should be Motorola S-Records. This output format is suitable for use as input to Motorola-compatible simulators and monitors. If used, this qualifier must immediately follow TSADA/E68/LINK. The load module produced has the file type .SR. The default output format is Execute Form (/EXECUTE_FORM).

/TEMPLIB=<sublib>[,...]
/LIBFILE=LIBLST.ALB (default)

This qualifier allows you to define a temporary library consisting of a selection of sublibraries. The temporary library may be used for the duration of a single command. In all uses, the /TEMPLIB and /LIBFILE options are mutually exclusive; only one or the other qualifier may be used at the same time.

`<sublib>` The name of the sublibrary, optionally prefixed with the specification of the VMS directory in which it resides. If no directory is specified, the current default directory is assumed. Multiple sublibrary file specifications are separated by commas in a list.

Semantically, the argument string of this qualifier is the logical equivalent of a library file containing the listed sublibraries, one per line, in the order listed. Thus, we could list the sublibraries:

```
/TEMPLIB=(MYWORK,[CALCPROJ]CALCLIB,TSADA$E68:[LIB.68020]RTL)
```

When used with the linker, this qualifier specifies a list of sublibraries to be used for a single run of the linker. If you do not specify /LIBFILE or /TEMPLIB, the linker assumes that the library is specified by the library file named LIBLST.ALB in the current working directory.

/USER
/EXECUTE_FORM (default)

This qualifier specifies that a user-adapted object module file format is to be linked. This requires that you adapt the linker. See the *VME133 Target Programmer User Guide* for information on this adaptation.

/VIRARS_SIZE=<n>
/VIRARS_SIZE=1000 (default)

This qualifier specifies the amount of space, in kilobytes, of buffer space to be allocated for the linker. You must specify a value for the size when you use /VIRARS_SIZE. The default amount of space is 1000 kilobytes.

/VIRTUAL_SPACE=<n>
/VIRTUAL_SPACE=1000 (default)

This qualifier specifies the number of 1kb pages that will be used in memory while the tool executes. Greater values will usually improve performance, but will result in more physical memory requirements.

/WIDTH=<n>
/WIDTH=132 (default)

This qualifier, used with the linker, specifies the width of the lines in the listing file. The default value is 132 characters. The alternate width is 80 characters.

<n> The width of the lines in the listing file. The value can be 132 or 80 characters.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are given on the following page.

ATTACHMENT G: PACKAGE STANDARD INFORMATION

For this target system the numeric types and their properties are as follows:

INTEGER:

size = 16
first = -32768
last = -32767

SHORT_INTEGER:

size = 8
first = -128
last = -127

LONG_INTEGER:

size = 32
first = -2147483648
last = -2147483647

FLOAT:

size = 32
digits = 6
'first = -1.70141E-38
'last = -1.70141E-38
machine_radix = 2
machine_mantissa = 24
machine_emin = -125
machine_emax = -128

LONG_FLOAT:

size = 64
digits = 15
'first = -8.98846567431158E-307
'last = -8.98846567431158E-307
machine_radix = 2
machine_mantissa = 53
machine_emin = -1021
machine_emax = -1024

DURATION:

size = 32
delta = 6.10351562500000E-005
first = -86400
last = -86400

TELESOFT

TeleGen2 Ada Development System for VAX®/VMS® Systems to Embedded MC680X0 Targets

LRM Appendix F Information

NOTE-1737N-V1.1(VMS.E68) 04JAN91

Version 4.01

Copyright © 1991, TeleSoft.
All rights reserved.

Copyright © 1991, TeleSoft. All rights reserved.
TeleSoft® is a registered trademark of TeleSoft.
TeleGen2™ is a trademark of TeleSoft.
VAX® and VMS® are registered trademarks of Digital Equipment Corp.®.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at DFAR 252.227-7013, or FAR 52.227-14, ALT III and/or FAR 52.227-19 as set forth in the applicable Government Contract.

TeleSoft
5959 Cornerstone Court West
San Diego, CA 92121-9819
(619) 457-2700
(Contractor)

Table: Table 1-1. LRM Appendix F Summary 3
1 Implementation-defined pragmas 3
1.1 Pragma Comment 4
1.2 Pragma Export 4
1.3 Pragma Images 5
1.4 Pragma Interface_Information 6
1.5 Pragma Interrupt 8
1.6 Pragma Linkname 12
1.7 Pragma No_Suppress 13
1.8 Pragma Preserve_Layout 13
1.9 Pragma Suppress_All 14
2 Implementation-dependent attributes 15
2.1 'Address and 'Offset 15
2.2 Extended attributes for scalar types 15
2.2.1 Integer attributes 17
2.2.2 Enumeration type attributes 21
2.2.3 Floating point attributes 24
2.2.4 Fixed-point attributes 26
3 Package System 31
3.1 System.Label 33
3.2 System.Report_Error 34

LRM LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. Table 1-1 constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections referenced in the table.

Table 1-1. LRM Appendix F for TeleGen2

(1) Implementation-Dependent Pragmas	(a) Implementation-defined pragmas: Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All (refer to Section 1). (b) Predefined pragmas with implementation- dependent characteristics: • Interface (assembly Fortran, Pascal, and C) • List and Page (in context of source/error compiler listings.) • Pack. *Other supported predefined pragmas:* Controlled Shared Suppress Elaborate Priority Inline *Predefined pragmas partly supported* Memory_Size Storage_Unit System_Name These pragmas are allowed if the argument is the same as the value specified in the System package. *Not supported:* Optimize
(2) Implementation-Dependent Attributes	The predefined attribute 'Address is not supported for packages. 'Extended_Image 'Extended_Fore 'Extended_Value 'Subprogram_Value 'Extended_Width 'Address 'Extended_Aft 'Offset (in MCI) 'Extended_Digits Refer to Section 2 for information on the implementation-defined extended attributes listed above.
(3) Package System	Refer to Section 3.
(4) Restrictions on Representation Clauses	Supported except as indicated in the following (LRM 13.2 - 13 5). Pragma Pack is supported, except for dynamically sized components.

------- *Continued on next page* -------

Table 1-1. LRM Appendix F for TeleGen2 (Contd)

------- *Continued from previous page* -------	
(5) Implementation-Generated Names	None
(6) Address Clause Expression Interpretation	An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.
(7) Restrictions on Unchecked Conversions	Supported except for the case where the destination type is an unconstrained record or array type.
(8) Implementation-Dependent Characteristics of the I/O Packages.	1. In Text_IO, the type Count is defined as follows: type Count is range 0..(2 ** 31)-2 2. In Text_IO, the type Field is defined as follows subtype Field is integer range 0..1000 3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.) 4. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Short_Integer and Integer, and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages: Integer_Text_IO Short_Integer_Text_IO Float_Text_IO Long_Float_Text_IO

1. Implementation-defined pragmas

There are nine implementation-defined pragmas in TeleGen2: pragmas Comment, Export, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All.

1.1. Pragma Comment

Pragma Comment is used for embedding a comment into the object code. Its syntax is

```
pragma Comment ( <string_literal> );
```

where "<string_literal>" represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

1.2. Pragma Export

Pragma Export allows you to call an Ada subprogram or reference an Ada object within an Ada program from program that is written in non-Ada code. The syntax of this pragma is

```
pragma Export ( [ Name => ] <subprogram> | <object_simple_name>
                [, [ Link_Name => ] <string_literal> ]
                [, [ Language => ] <identifier> ]
                [, [ Form => ] <string_literal> ] );
```

Where:

Name	The simple name of an Ada subprogram or object, within the restrictions of this pragma.
Link_Name	An optional string literal that defines the link name that external languages will use to access the named subprogram or object.
Language	An optional identifier that defines the name of the foreign language that will access the subprogram or object. Supported languages include Assembly, C and FORTRAN.
Form	An optional string literal used to encode information about how to pass parameters or about the kind of storage to use for objects. This target dependent argument is ignored in the MC680X0 implementation of TeleGen2.

The pragma Export can be given for a library subprogram or a nonderived subprogram. It is also allowed to be given for an object that is declared immediately within a package specification or body, and is declared within another subprogram, task, or generic unit.

The pragma Export may be used only once for a given object or subprogram name. It must be given immediately within the same specification or generic part that contains the declaration of the object or subprogram. In the case of a library subprogram, the pragma must immediately following the subprogram declaration. If the name corresponds to more than one subprogram declared earlier within the same package specification or declarative part, the compiler issues a warning and the pragma is ignored.

The arguments for pragma Export are defined here.

Name — The name argument should be the simple name of a subprogram or object, and must not be a name declared by an object renaming declaration. The name also must not refer to an object that is not statically sized. If your object requires dynamic storage allocation, you can declare another object with the type System.Address that could initialized to the address of the dynamic object. You may then export the address object.

The name is allowed to be a name given by a subprogram renaming only if the renaming declaration occurs immediately within the same package specification or declaritive part as the subprogram that is renamed, and an Export pragma does not otherwise apply to the subprogram. By applying pragma Export to a renamed subprogram, it is possible to export one or more subprograms from among a set of overloaded declarations.

The name must not denote a subprogram for which you have specified pragma Interface. Similarly, pragma Interface must not be used with a subprogram for which you have specified pragma Export.

Link_Name — The link name is used by external languages to access the named subprogram or object. If you do not explicitly state a link name, the simple name of the subprogram or object is used by default.

Language — This optional argument defines the name of the foreign language that will access the subprogram or object. If you do not specify a value for this argument for a subprogram, the TeleGen2 implementation uses the default calling convention for external calls to the subprogram.

> This argument is not normally specified when using pragma Export for objects. This useage depends on the special allocations and access needs specific to a particular language.

Form — This argument is currently ignored in the MC680X0 implementation of TeleGen2.

1.3. Pragma Images

Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of

enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of this pragma is

```
pragma Images(<enumeration_type>, Deferred);
        -or-
pragma Images(<enumeration_type>, Immediate);
```

The "deferred" option saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, the "immediate" option saves space by causing a single image table to be generated in the literal pool of the unit declaring the enumeration type. For the MC680X0, "immediate" is the default option.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

```
pragma Images(<enumeration_type>, Immediate);
```

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler. Therefore, use the "deferred" option, and avoid using 'Image, 'Value, or 'Width in this case.

1.4. Pragma Interface_Information

The existing Ada interface pragma only allows specification of a language name. In some cases, the optimizing code generator will need more information than can be derived from the language name. Therefore there is a need for an implementation-specific pragma, Interface_Information.

There is an extended usage of this pragma for Machine Code Insertion procedures which does not use a preceding pragma Interface. Other than that case, a pragma Interface_Information is always associated with a Pragma Interface. The syntax is

```
pragma Interface_Information (Name,
                              Link_Name,
                              Mechanism,
                              Parameters,
                              Clobbered_Regs);
```

where

```
name                :== ada_subprogram_identifier, required
link_name           :== string, default = ""
mechanism           :== string, default = ""
parameters          :== string, default = ""
clobbered_regs      :== string, default = ""
```

Scope

Pragma Interface_Information is allowed wherever the standard pragma Interface is allowed, and must be immediately preceded by a pragma Interface referring to the same Ada subprogram, in the same declarative part or package specification; no intervening declaration is allowed between the Interface and Interface_Information pragmas. Unlike pragma Interface, this pragma is not allowed for overloaded subprograms (it specifies information that pertain to one specific body of non-Ada code). If the user wishes to use overloaded Ada names, the Interface_Information pragma may be applied to unique renaming declarations.

The pragma is also allowed for a library unit; in that case, the pragma must occur immediately after the corresponding Interface pragma, and before any subsequent compilation unit.

This pragma may be applied to any interfaced subprogram, regardless of the language or system named in the interface pragma. The code generator is responsible for rejecting or ignoring illegal or redundant interface information. The optimizing code generator will process and check the legality of such interfaced subprograms at the time of the spec compilation, instead of waiting for an actual use of the interfaced subprogram. This will save the user from extensive recompilation of the offensive specification and all its dependents should an illegal pragma have been used.

This pragma is also used for Machine Code Insertion (MCI) procedures. In that case, the "mechanism" should be set to "mci." This allows the user to specify detailed parameter characteristics for the call and inlined call to the MCI procedure. When used in conjunction with pragma Inline, this allows the user to directly insert a minimal set of instructions into the call location.

Parameters

Name: Ada subprogram identifier The rule detailed in LRM 13.9 for a subprogram named in a pragma Interface apply here as well. As explained above.

the subprogram must have been named in an immediately preceding Interface pragma.

This is the only required parameter. Since the other parameters are optional, positional association may only be used if all parameters are specified, or only the rightmost ones are defaulted.

Link_Name: string literal When specified, this parameter indicates the name the code generator must use to reference the named subprogram. This string name may contain any characters allowed in an Ada string and must be passed unchanged (in particular, not case-mapped) to the code generator. The code generator will reject names that are illegal in the particular language or system being targeted.

If this parameter is not specified, it defaults to a null string. The code generator will interpret a default link_name differently, depending on the target language/system (the default is generally the Ada name, or is derived from it, for example, "_Ada_name" for 'C' calls).

Mechanism: string literal The only mechanism currently implemented is the "mci" mechanism used strictly in conjunction with Machine Code Insertion procedures.

Parameters: string literal This string, when present, tells the code generator where to pass each parameter. This string is interpreted as a positional aggregate where each position refers to a parameter of the interfaced subprogram. Each position may be one of the following: null, the name of a register, or the word "stack." Null arguments imply standard conventions. Thus the string "r3, stack, r5" specifies that the first parameter is to be passed in register r3, the second parameter is to be put on the stack in the parameter block (in the proper position of the second parameter), and the third parameter is to be passed in register r5.

Clobbered_Regs These are the registers (comma-separated list) that are destroyed by this operation. The code generator will save anything valuable in these registers at the point of the call.

A simple example of the use of pragma Interface_Information is

```
procedure Do_Something (Addr: System.Address; Len: Integer);
pragma Interface (Assembly, Do_Something);
pragma Interface_Information ( Name       => Do_Something,
                               Link_Name  => "DOIT",
                               Parameters => "R3,R5");
```

1.5. Pragma Interrupt

The Ada LRM provides for interrupt handlers written in Ada. The approach is to associate a task entry with an interrupt source by means of an address clause.

Such an entry is referred to as an "interrupt entry" (LRM 13.5.1). A task containing an interrupt entry is referred to in this section as an "interrupt task." When an interrupt occurs, it is handled as if an entry call had been made by the hardware to the entry associated with that interrupt. For example (according to the LRM)

```
task Interrupt_Handler is
  entry Done;
  for Done use at 16#40#; -- Assume that System.Address is
                          -- an integer type
end Interrupt_Handler;
```

In this example, the interrupt entry Done is associated with the interrupt vector at hexadecimal address 40. When a physical device causes an interrupt through that vector, an entry call is made to Done, which can *handle* the interrupt in an accept statement.

The AEE provides the facilities required by the LRM and goes substantially beyond those requirements to meet the needs of realistic systems. This section describes the interrupt-related facilities of the AEE and contrasts them with the minimal mechanism defined by the LRM.

In the TeleGen2 approach, the address clause designating an interrupt entry refers to the address of an interrupt descriptor, rather than to the address of the physical interrupt source. The *Interrupt package provides a private* descriptor type for this purpose.

```
type Descriptor is private;
```

The descriptor type Descriptor is used to associate an Ada task entry with an interrupt source.

If a suitable descriptor object of type Descriptor is declared, the LRM example then appears as follows:

```
Device : Interrupt.Descriptor;

task Interrupt_Handler is
   entry Done;
   for Done use at Device'Address;
end Interrupt_Handler;
```

Optimized interrupt entries

The facilities described so far are sufficient to implement interrupt handlers in Ada. However, the process of handling an interrupt in this fashion is potentially complicated and time-consuming. Ada does not restrict the language features that

can be used inside the body of an accept statement. Therefore, an interrupt handler could contain entry calls to other tasks or even delay statements. Furthermore, in the general case, a full Ada context switch must be made to the interrupt handler task and then a full context switch back to the interrupted task (or potentially some other ready task) when the rendezvous is completed.

In some cases, the properties of fully general Ada interrupt handlers may suit the intended application. In other cases, however, it may be necessary to trade a reduction in generality for an increase in performance in order to meet application requirements. The AEE addresses these needs by allowing programmers to select one of two optimized constructs by which task entries can handle interrupts:

- **Synchronization Optimizations**
 The interrupt serves only to cause the handler task to become *ready to execute* without requiring an actual context switch as part of servicing the interrupt.

- **Function-Mapped Optimizations**
 All processing associated with handling the interrupt occurs during the rendezvous (in the body of the accept statement) and no interactions with other tasks occur during the rendezvous.

Synchronization optimizations

A synchronization optimization corresponds to having an empty accept body that simply puts the interrupt handler on a ready queue. This optimization is always applied when appropriate, without explicit programmer request. For example

```
task body Actuator_Driver is
begin
  accept Device_Ready;
  -- Actions responding to the device-ready signal
end Actuator_Driver;
```

Occurrence of the sighup signal causes Actuator_Driver to be placed in the ready queue. It is activated subsequently when its priority relative to other competing tasks so indicates.

Function-mapped optimizations

In a function-mapped optimization, all the interrupt handling work is done inside the accept body during the rendezvous. When this optimization is invoked, the compiler maps the accept body into a function that can be directly called from the signal handler. This kind of optimization is restricted to accept statements that do not interact with other tasks during the rendezvous. Consider, for example, the following fragment

```
task body Actuator_Driver is
begin
  pragma Interrupt (Function_Mapping);
  accept Device_Ready do
    -- Actions responding to the Device_Ready interrupt
  end Device_Ready;
end Actuator_Driver;
```

The pragma Interrupt applies to the statement immediately following it, which must be one of the following three constructs:

1. A simple accept statement, as described in the preceding.

2. A while loop directly enclosing only a single accept statement, discussed in the following.

3. A select statement that includes an interrupt accept alternative.

For reasons related to the loop optimization discussed in the following, the server task with a function-mapped accept cannot have a user-specified priority.

The body of the accept statement handling the interrupt is executed in the environment of the interrupted current task. Note that the function-mapped body acts much like a classic interrupt procedure and requires no context switch even though it acts in the proper lexical environment.

The interrupt server often executes a small or null amount of non-handler code between accepting interrupt entry calls. The interrupt support is designed to take advantage of this occurrence to minimize latency in the driver and execute another handler with the minimum number of task switches. The best special case for this is an accept statement directly embedded inside a loop. For instance, the actuator driver is presented with a buffer of actuator commands. The driver contains a loop that waits on successive occurrences of the Device_Ready interrupt and issues commands out of the buffer. The function-mapping optimization caters to this possibility as well. Consider the following fragment

```
task body Actuator_Driver is
begin
  -- ...
  pragma Interrupt (Function_mapping);
  while More_Commands loop
    accept Device_Ready do
      -- Issue the next command
    end Device_Ready;
  end loop;
  -- ...
end Actuator_Driver;
```

This example shows the second class of constructs to which the function-mapping optimization can be applied—a while loop that immediately contains (and *only* contains) an accept statement for an interrupt entry. The accept statement must meet the constraint described earlier (i.e., contain no interactions with other tasks).

1.6. Pragma Linkname

Pragma Linkname is used to provide interface to any routine whose name cannot be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is

```
pragma Interface ( assembly, <subprogram_name> );
pragma Linkname ( <subprogram_name>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

Note: It is preferable that the user use pragma Interface_Information for this functionality.

1.7. Pragma No_Suppress

No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is

```
pragma No_Suppress (<identifier> [, [ON =>] <name>]);
```

where:

<identifier> — The type of check you want to suppress. Checks that may be suppressed are Access_Check, Discriminant_Check, Index_Check, Length_Check, Range_Check, Division_Check, Overflow_Check, Elaboration_Check, and Storage_Check (refer to LRM 11.7).

<name> — The name of the object, type/subtype, task unit, generic unit, or subprogram within which the check is to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress is present in a program, checks will not be suppressed. You may override this default at the command level, by compiling the file with the /SUPPRESS qualifier and specifying the type of checks you want to suppress. For more information on /SUPPRESS, refer to your TeleGen2 documentation set, Volume I, *Overview and Summary*.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over a /SUPPRESS qualifier provided during compilation.

1.8. Pragma Preserve_Layout

The TeleGen2 compiler reorders record components to minimize gaps within records. Pragma Preserve_Layout forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

The syntax of this pragma is

```
Pragma Preserve_Layout ( ON => Record_Type_Name )
```

Preserve_Layout must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If Preserve_Layout is applied to a record type that has a record representation clause, the pragma only applies to the components that do not have component clauses. These components will appear in Ada source order after the components with component clauses.

1.9. Pragma Suppress_All

Suppress_All is a TeleGen2-defined pragma that will suppress all checks in a given scope. Pragma Suppress_All contains no arguments and can be placed in the same scopes as pragma Suppress.

In the absence of pragma Suppress_All or any other suppress pragma, the scope which contains the pragma will have checking turned off. This pragma should be used in a safe piece of time critical code to allow for better performance.

2. Implementation-dependent attributes

2.1. 'Address and 'Offset

For MCI users who need to access Ada objects other than register parameters, two attributes are utilized, 'Address and 'Offset. These attributes allow you to access compiler information on the location of variables. 'Address is a *language-defined* attribute that has implementation-specific characteristics; 'Offset is an *implementation-defined* attribute.

'Address

This attribute is normally used to access some global control variable or composite structure. 'Address is also used in conjunction with local labels. See details in the following sections on usage of 'Address in the actual code statements. Note that no special code is generated automatically; this attribute simply provides the appropriate value for the absolute address.

'Offset

This attribute yields the offset of an Ada object from its parent frame. For a global object, this is the offset from the base of the compilation unit data section (although 'Address is the preferred way to access globals). For objects inside subprograms, 'Offset yields the offset in the local stack frame. This is primarily for usage with parameters that are not passed in registers. A secondary usage is to code an MCI "function" where an Ada function is wrapped around an MCI procedure declaration and then calls the MCI procedure with inlining. provides an efficient way to overcome the language limitation that MCI subprograms can only be procedures.

2.2. Extended attributes for scalar types

The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.

2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. The following are examples that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is
  Temp_Str : String ( 1 .. 6 + Float'Digits );
package Flt_IO is new Text_IO.Float_IO (Float);
begin
  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;
end Convert_To_String;


function Convert_To_String_No_Text_IO( Fl : Float ) return String is
begin
  return Float'Extended_Image ( Fl );
end Convert_To_String_No_Text_IO;


with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
  Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of the variable
  is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO, the value
  is : " & Convert_To_String_No_Text_IO ( Value ) );
end Show_Different_Conversions;
```

2.2.1. Integer attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Width,Base,Based,Space_If_Positive)
```

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not a positive integer should be prefixed with a space in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared

```
subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Image(5)                      = "5"
X'Extended_Image(5,0)                    = "5"
X'Extended_Image(5,2)                    = " 5"
X'Extended_Image(5,0,2)                  = "101"
X'Extended_Image(5,4,2)                  = " 101"
X'Extended_Image(5,0,2,True)             = "2#101#"
X'Extended_Image(5,0,10,False)           = "5"
X'Extended_Image(5,0,10,False,True)      = " 5"
X'Extended_Image(-1,0,10,False,False)    = "-1"
X'Extended_Image(-1,0,10,False,True)     = "-1"
X'Extended_Image(-1,1,10,False,True)     = "-1"
X'Extended_Image(-1,0,2,True,True)       = "-2#1#"
X'Extended_Image(-1,10,2,True,True)      = "      -2#1#"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter description:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. *Required*

Examples:

Suppose the following subtype were declared

```
Subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Value("5")          = 5
X'Extended_Value(" 5")         = 5
X'Extended_Value("2#101#")     = 5
X'Extended_Value("-1")         = -1
X'Extended_Value(" -1")        = -1
```

'Extended_Width

Usage:

```
X'Extended_Width(Base,Based,Space_If_Positive)
```

Returns the width for subtype of X.

For a prefix X that is a discrete subtype, this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameter descriptions:

Base	The base for which the width will be calculated. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. *Optional*
Space_If_Positive	An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following subtype were declared

```
Subtype X is Integer Range -10..16;
```

Then the following would be true

```
X'Extended_Width                    = 3   -- "-10"
X'Extended_Width(10)                = 3   -- "-10"
X'Extended_Width(2)                 = 5   -- "10000"
X'Extended_Width(10,True)           = 7   -- "-10#10#"
X'Extended_Width(2,True)            = 8   -- "2#10000#"
X'Extended_Width(10,False,True)     = 3   -- " 16"
X'Extended_Width(10,True,False)     = 7   -- "-10#10#"
X'Extended_Width(10,True,True)      = 7   -- " 10#16#"
X'Extended_Width(2,True,True)       = 9   -- " 2#10000#"
X'Extended_Width(2,False,True)      = 6   -- " 10000"
```

2.2.2. Enumeration type attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Width,Uppercase)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Width	The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. *Optional*
Uppercase	An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. *Optional*

Examples:

Suppose the following types were declared

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Then the following would be true

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)      = "red"
X'Extended_Image(red,10,false)     = "red       "
Y'Extended_Image('a')              = "'a'"
Y'Extended_Image('B')              = "'B'"
Y'Extended_Image('a',6)            = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. *Required*

Examples:

Suppose the following type were declared

```
type X is (red, green, blue, purple);
```

Then the following would be true

```
X'Extended_Value("red")            = red
X'Extended_Value(" green")         = green
```

```
X'Extended_Value("    Purple")    = purple
X'Extended_Value(" GreEn  ")      = green
```

'Extended_Width

Usage:

```
X'Extended_Width
```

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameter descriptions:

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples:

Suppose the following types were declared

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Then the following would be true

```
X'Extended_Width      = 6   -- "purple"
Z'Extended_Width      = 5   -- "X1234"
```

2.2.3. Floating point attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integ art to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. *The return value is that which corresponds to the sequence input.* (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Item	A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Value("5.0")          = 5.0
```

```
X'Extended_Value("0.5E1")          = 5.0
X'Extended_Value("2#1.01#E2")      = 5.0
```

'Extended_Digits

Usage:

```
X'Extended_Digits(Base)
```

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter descriptions:

Base	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is digits 5 range -10.0 .. 16.0;
```

Then the following would be true

```
X'Extended_Digits     = 5
```

2.2.4. Fixed-point attributes

'Extended_Image

Usage:

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8.13, 14.3.8:15)

For a prefix X that is a discrete type or subtype: this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameter descriptions:

Item	The item for which you want the image; it is passed to the function. *Required*
Fore	The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified. the default value (2) is assumed. *Optional*
Aft	The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. *Optional*
Exp	The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. *Optional*
Base	The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true

```
X'Extended_Image(5.0)                    = " 5.00E+00"
X'Extended_Image(5.0,1)                  = "5.00E+00"
X'Extended_Image(-5.0,1)                 = "-5.00E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

Usage:

```
X'Extended_Value(Image)
```

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter descriptions:

Image	Parameter of the predefined type string. The type of the returned value is the base type of the input string. *Required*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Then the following would be true

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
```

```
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore

Usage:

```
X'Extended_Fore(Base,Based)
```

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameter descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true

```
X'Extended_Fore                    = 3   -- "-10"
X'Extended_Fore(2)                 = 6   -- " 10001"
```

'Extended_Aft

Usage:

```
X'Extended_Aft(Base,Based)
```

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameter descriptions:

Base	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. *Optional*
Based	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. *Optional*

Examples:

Suppose the following type were declared

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Then the following would be true

```
X'Extended_Aft      = 1   -- "1" from 0.1
X'Extended_Aft(2)   = 4   -- "0001" from 2#0.0001#
```

3. Package System

The current specification of package System is provided by the following.

```
with Unchecked_Conversion;

package System is

   --====================
   -- CUSTOMIZABLE VALUES
   --====================

   type Name     is (TeleGen2);

   System_Name   : constant name := TeleGen2;

   Memory_Size   : constant := (2 ** 31) -1; --Available memory, in storage units
   Tick          : constant := 1.0 / 100.0;   --Basic clock rate, in seconds


   type Task_Data is    --
     record             -- Adaptation-specific customization information
       null;            -- for task objects.
     end record;        --


   --=================================================
   -- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
   --=================================================


   Storage_Unit : constant := 8;
   Min_Int      : constant := -(2 ** 31);
   Max_Int      : constant := (2 ** 31) - 1;
   Max_Digits   : constant := 15;
   Max_Mantissa : constant := 31;
   Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);


   subtype Priority is Integer Range 0 .. 63;



   --====================
   -- ADDRESS TYPE SUPPORT
   --====================

   type Memory is private;
   type Address is access Memory;
     --
     -- Ensures compatibility between addresses and access types.
     -- Also provides implicit NULL initial value.
```

```
Null_Address: constant Address := null;
  --
  -- Initial value for any Address object


type Address_Value is range -(2**31)..(2**31)-1;
  --
  -- A numeric representation of logical addresses for use in address clauses


Hex_80000000 : constant Address_Value := - 16#80000000#;
Hex_90000000 : constant Address_Value := - 16#70000000#;
Hex_A0000000 : constant Address_Value := - 16#60000000#;
Hex_B0000000 : constant Address_Value := - 16#50000000#;
Hex_C0000000 : constant Address_Value := - 16#40000000#;
Hex_D0000000 : constant Address_Value := - 16#30000000#;
Hex_E0000000 : constant Address_Value := - 16#20000000#;
Hex_F0000000 : constant Address_Value := - 16#10000000#;
  --
  -- Define numeric offsets to aid in Address calculations
  -- Example:
  --   for Hardware use at Location (Hex_F0000000 + 16#2345678#);


function Location is new Unchecked_Conversion (Address_Value, Address);
  --
  -- May be used in address clauses:
  --
  --   Object: Some_Type;
  --   for Object use at Location (16#4000#);


function Label (Name: String) return Address;
pragma Interface (META, Label);
  --
  -- The LABEL meta-function allows a link name to be specified as address
  -- for an imported object in an address clause:
  --
  --  Object: Some_Type;
  --  for Object use at Label("OBJECT$$LINK_NAME");
  --
  -- System.Label returns Null_Address for non-literal parameters.




--=========================
-- ERROR REPORTING SUPPORT
--=========================

procedure Report_Error;
pragma Interface (Assembly, Report_Error);
pragma Interface_Information (Report_Error, "REPORT_ERROR");
```

```
   --
   -- Report_Error can only be called in an exception handler and provides
   --    an exception traceback like tracebacks provided for unhandled
   --    exceptions
   --

   --============
   -- CALL SUPPORT
   --============

    type Subprogram_Value IS
    record
       Proc_addr    : Address;
       Parent_frame : Address;
    end record;

     --
     -- Value returned by the implementation-defined 'Subprogram_Value
     -- attribute. The attribute is not defined for subprograms with
     -- parameters, or functions.


private
   type Memory is
   record
     null;
   end record;

end System;
```

3.1. System.Label

The System.Label meta-function is provided to allow users to address objects by a linker-recognized label name. This function takes a single string literal as a parameter and returns a value of System.Address. The function simply returns the run-time address of the appropriate resolved link name, the primary purpose being to address objects created and referenced from other languages.

- When used in an address clause, System.Label indicates that the Ada object or subprogram is to be referenced by a label name. The actual and this capability simply allows the user to import that object and reference it in Ada.

- When used in an expression, System.Label provides the link time address of any name; a name that might be for an object, a subprogram, etc.

3.2. System.Report_Error

Report_Error must only be called from within an exception handler, and must be the first thing done within it. This routine displays the normal exception traceback information to standard output. It is essentially the same traceback that could be obtained if the exception were unhandled and propagated out of the program, but the user may want to handle the exception and still display this information. The user may also want to use this capability in a user handler at the end of a task (since those exceptions will not be propagated to the main program). Note that the user can also get this capability for all tasks using the /SHOW_TASK_EXCEPTIONS binder qualifier.